Fish & Richardson P.C.

Frederick P. Fish
1855-1930

W.K. Richardson
1859-1951

2200 Sand Hill Road
Suite 100
Menlo Park, California
94025

Telephone
650 322-5070

Facsimile
650 854-0875

Web Site
www.fr.com

BOSTON
NEW YORK
SILICON VALLEY
SOUTHERN CALIFORNIA
TWIN CITIES
WASHINGTON, DC

February 2, 1999

Attorney Docket No.: 08993/007001

Box Patent Application
Assistant Commissioner for Patents
Washington, DC 20231

Presented for filing is a new original patent application of:

Applicant:	JOSHUA B. SUSSER AND JUDITH E. SCHWABE
Title:	OBJECT-ORIENTED INSTRUCTION SET FOR RESOURCE-CONSTRAINED DEVICES

Enclosed are the following papers, including all those required to receive a filing date under 37 CFR §1.53(b):

	Pages
Specification	22
Claims	9
Abstract	1
Declaration	1
Drawing(s)	14

Enclosures:

- Assignment cover sheet and an Assignment, 2 pages, and a separate $40.00 fee
- Appendix cover sheet (1 page) and Java Card Virtual Machine Specification (118 pages).
- Postcard.

Basic filing fee	$ 760.00
Total claims in excess of 20 times $18.00	684.00
Independent claims in excess of 3 times $78.00	390.00
Multiple dependent claims	0.00
Total filing fee:	$ 1834.00

"EXPRESS MAIL" Mailing Label Number TB 888 891 432 US

Date of Deposit February 2, 1999

A check for the filing fee is enclosed. Please apply any other required fees or any credits to deposit account 06-1050, referencing the attorney docket number shown above.

If this application is found to be INCOMPLETE, or if a telephone conference would otherwise be helpful, please call the undersigned at 650/322-5070.

Kindly acknowledge receipt of this application by returning the enclosed postcard.

Please send all correspondence to:

Wayne P. Sobon
Fish & Richardson P.C.
2200 Sand Hill Road, Suite 100
Menlo Park, CA 94025

Respectfully submitted,

Wayne P. Sobon
Reg. No. 32,438

Enclosures
103064.PAL1

APPLICATION

FOR

UNITED STATES LETTERS PATENT

TITLE: OBJECT-ORIENTED INSTRUCTION SET FOR RESOURCE-CONSTRAINED DEVICES

APPLICANT: JOSHUA B. SUSSER AND JUDITH E. SCHWABE

"EXPRESS MAIL" Mailing Label Number TB 888 891 432 US

Date of Deposit February 2, 1999

OBJECT-ORIENTED INSTRUCTION SET FOR RESOURCE-CONSTRAINED DEVICES

CROSS REFERENCE TO RELATED APPLICATIONS

The following applications are incorporated herein by reference in their entirety:

"Architecture-Neutral Exception Handling" and "Token-Based Linking," each naming Joshua B. Susser and Judith E. Schwabe as inventors, which are being filed concurrently with the present application; and

"Virtual Machine with Securely Distributed Bytecode Verification," naming Moshe Levy and Judy Schwabe as inventors, filed April 15, 1997.

In addition, an Appendix entitled "Java Card™ Virtual Machine Specification: Java Card Version 2.1" is attached to this application and forms a part of the present specification.

BACKGROUND

The present invention relates, in general, to object-oriented, architecture-neutral programs for use with

resource-constrained devices such as smart cards and the like.

A virtual machine is an abstract computing machine generated by a software application or sequence of instructions which is executed by a processor. The term "architecture-neutral" refers to programs, such as those written in the Java™ programming language, which can be executed by a virtual machine on a variety of computer platforms having a variety of different computer architectures. Thus, for example, a virtual machine being executed on a Windows™-based personal computer system will use the same set of instructions as a virtual machine being executed on a UNIX™-based computer system. The result of the platform-independent coding of a virtual machine's sequence of instructions is a stream of one or more bytecodes, each of which is, for example, a one-byte-long numerical code.

Use of the Java programming language has found many applications including, for example, those associated with Web browsers.

The Java programming language is object-oriented. In an object-oriented system, a "class" describes a collection of data and methods that operate on that data. Taken together, the data and methods describe the state of and behavior of an object.

Java also is verifiable such that, prior to execution of an application written in the Java programming language, a determination can be made as to whether any instruction sequence in the program will attempt to process data of an improper type for that bytecode or whether execution of bytecode instructions in the program will cause underflow or overflow of an operand stack.

A Java™ virtual machine executes virtual machine code written in the Java programming language and is designed for use with a 32-bit architecture. However, various resource-constrained devices, such as smart cards, have an 8-bit or 16-bit architecture.

Smart cards, also known as intelligent portable data-carrying cards, generally are made of plastic or metal and have an electronic chip that includes an embedded microprocessor to execute programs and memory to store programs and data. Such devices, which can be about the size of a credit card, typically have limited memory capacity. For example, some smart cards have less than one kilo-byte (1K) of random access memory (RAM) as well as limited read only memory (ROM), and/or non-volatile memory such as electrically erasable programmable read only memory (EEPROM). The limited architecture and memory make it impractical or impossible to implement the full Java Virtual Machine on the device.

Furthermore, smart cards come with a variety of processors and configurations. Thus, it is desirable to provide a platform-independent programming language that can be executed on such a resource-constrained device.

SUMMARY

In general, a verifiable, object-based, type-safe and pointer-safe instruction set is described for application software programs which can be downloaded to and executed on a range of resource-constrained devices.

According to one aspect, an application software program includes an object-oriented, verifiable, type-safe and pointer-safe sequence of instructions residing on a computer-readable medium. The program can be loaded to and executed by a resource-constrained device that is based on

an architecture of fewer than 32 bits, such as a 16-bit or 8-bit architecture.

According to another aspect, an application software program includes an object-oriented, verifiable, type-safe and pointer-safe sequence of instructions residing on a computer-readable medium. The program can be loaded to and executed by a resource-constrained device having random access memory with a capacity of no more than about 64K.

Various implementations include one or more of the following features. For example, each instruction can include an 8-bit operation code, and the sequence of instructions can be hardware platform-independent. In some implementations, the sequence includes instructions that were previously converted from at least one Java class file with at least some references to a constant pool transformed to inline data. For example, the instructions can include operation codes and operands. Some references to the constant pool can be inlined into operands, and some references to the constant pool can be inlined into operation codes.

Similarly, in some embodiments, the instructions can be executed by a device that supports multiple data types. The sequence of instructions can include data manipulation instructions each of which is specific to a particular data type. In some implementations, the data type associated with each data manipulation instruction is selected from among one of the following types: an 8-bit signed two's complement integer numeric type, a 16-bit signed two's complement integer numeric type and a 32-bit signed two's complement integer numeric type. Additionally, the instructions can be executed by a device that supports multiple reference types each of which is selected from among one of the following types: a class type, an interface

type and an array type. Furthermore, the program can include one or more composite instructions for performing an operation on a current object.

According to another aspect, a resource-constrained device includes memory for storing an application software program comprising an object-oriented, verifiable, type-safe and pointer-safe sequence of instructions. The device also includes a virtual machine implemented on a microprocessor. The virtual machine is capable of executing the sequence of instructions. In various embodiments, the device may be based on a limited architecture or may have a limited amount of memory. For example, in some implementations, the device includes random access memory having a capacity of no more than about 64K. In other embodiments, the microprocessor is based on an architecture of less than 32 bits, for example, a 16 or 8-bit architecture.

In other embodiments, an application-specific integrated circuit (ASIC) or a combination of a hardware and firmware can be used instead of a virtual machine running on a microprocessor.

In one particular application of the invention, the resource-constrained device is a smart card. The smart card can include a virtual machine implemented on a microprocessor, wherein the virtual machine is capable of executing a sequence of instructions such as those described above.

According to another aspect, methods are disclosed for using an application software program including an object-oriented, verifiable, type-safe and pointer-safe sequence of instructions. The software program can be received in a resource-constrained device having, for example, either limited memory or based on a limited architecture. The sequence of instructions then can be

executed on the resource-constrained device. In some implementations, the software program can be accessed over a computer network such as the Internet prior to downloading it onto the device. When the program is downloaded to the resource-constrained device, constant pool indices that appear in the received set of instructions can be transformed to corresponding data values.

Various implementations include one or more of the following advantages. By supporting many, although not all, of the features of the Java language and by using the same semantics as the Java class files, platform-independent virtual machine code can be written to be executed by a smart card or other resource-constrained device.

The instruction set can inline certain data, which would otherwise appear as part of a constant pool, directly into operation codes or operands. Thus, the instruction set itself can incorporate certain information that would otherwise be stored in and obtained from a constant pool if one were using the Java class file format. By inlining some of the information directly into the instruction set, the size of the constant pool can be reduced, which can help reduce the amount of memory required to store the constant pool and can improve the execution speed of the bytecode. In some cases, inlining the information directly into an operation code can reduce the number of operands required for a particular instruction. Further inlining of information from a constant pool when the program is downloaded to the resource-constrained device can either eliminate the need to retain the constant pool on the device or reduce the size of the constant pool.

Other features such as composite instructions for performing operations on the current object and the explicit

handling of 16-bit arithmetic can further reduce the length of a bytecode program.

Other features and advantages will be readily apparent from the following detailed description, the accompanying drawings and the claims.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 illustrates an exemplary system including a virtual machine residing on a smart card according to the invention.

FIG. 2 is a flow chart illustrating a method of providing executable code to a smart card according to the invention.

FIGS. 3A and 3B illustrate, respectively, an exemplary format of virtual machine instruction and an inner loop of execution of the virtual machine according to the invention.

FIGS. 4A and 4B are tables of an exemplary set of operation codes for the virtual machine listed in numerical order by operation code and in alphabetical order by mnemonic, respectively.

FIG. 5 is a list of data types which are supported by operation codes that exist for multiple data types according to the invention.

FIG. 6A illustrates the format of an "iipush" instruction according to the invention, and

FIG. 6B illustrates the format of a corresponding "ldc" instruction in the Java class file format.

FIG. 7A illustrates the format of a "checkcast" instruction in the Java class file format, and

FIG. 7B illustrates the format of a "checkcast" instruction according to the invention.

FIG. 8A illustrates the format of a family of "getfield_T" instructions according to the invention, and

FIG. 8B illustrates the format of a corresponding "getfield" instruction in the Java class file format.

FIG. 9A and 9B illustrate how an implementation program on the smart card prepares virtual machine code for installation on the smart card according to one embodiment of the invention.

FIGS. 10A and 10B illustrate alternative instructions for obtaining the same result according to the invention.

FIG. 11A illustrates bytecodes for carrying out a mathematical expression using the Java class file format, and

FIG. 11B illustrates bytecodes for carrying out the same mathematical expression according to the invention.

FIG. 12 is partial, non-exclusive list of resource-constrained devices with which the invention can be used.

DESCRIPTION

A verifiable, object-based, type-safe and pointer-safe instruction set is described below for application software programs which can be downloaded to and executed on a range of resource-constrained devices. Resource-constrained devices are generally considered to be those that are relatively restricted in memory and/or computing power or speed, as compared to conventional desktop computers and the like. Although the particular implementation discussed below is described in reference to a smart card, the invention can be used with other resource-constrained devices including, but not limited to, cellular telephones, boundary scan devices, field programmable

devices, personal digital assistants (PDAs) and pagers, as well as other miniature or small footprint devices.

Programs written with the instruction set described below are capable of being downloaded to and executed on resource-constrained devices having about sixty-four kilo-bytes (64K) of RAM or less. Some of the resource-constrained devices in which such programs can be executed may have no more than about sixteen kilo-bytes (16K) of RAM and others may have no more than about four kilo-bytes (4K) of RAM. Many of the devices also have limited amounts of other memory, such as no more than about twenty-four kilo-bytes (24K) of ROM, or no more than about 16K of non-volatile memory such as EEPROM. Similarly, some resource-constrained devices are based on an architecture designed for fewer than 32 bits. For example, some of the devices which can be used with the invention are based on an 8-bit or 16-bit architecture, rather than a 32-bit architecture. Of course, applications using the instruction set described below are upward compatible and can be executed, for example, on other Java platforms provided equivalent device support is present.

Referring to FIGS. 1 and 2, development of an applet for a resource-constrained device, such as a smart card 40, begins in a manner similar to development of other Java programs. In other words, a developer writes one or more JAVA classes (step 60) and compiles the source code with a JAVA compiler to produce one or more class files 10 (step 62). The applet can be run, tested and debugged, for example, on a workstation using simulation tools to emulate the environment on the card 40. When the applet is ready to be downloaded to the card 40, the class files 10 are converted to a converted applet (CAP) file 16 by a converter 14 (step 64). The converter 14 can be implemented as a Java

application being executed by a desktop computer. The converter 14 can accept as its input one or more export files 12 in addition to the class files 10 to be converted. An export file 12 contains naming or linking information for the contents of other packages that are imported by the classes being converted.

In general, the CAP file 16 includes all the classes and interfaces defined in a single Java package and is represented by a stream of 8-bit bytes. All 16-bit and 32-bit quantities are constructed by reading in two or four consecutive 8-bit bytes, respectively. Among other things, the CAP file 16 includes a constant pool component 18 which is packaged separately from a method component 20. The constant pool component 18 can include various types of constants, ranging from numerical literals known at compile time to method and field references which are resolved either when the program is downloaded to the smart card 40 or at the time of execution by the smart card. The method component 20 specifies the set of instructions to be downloaded to the smart card 40 and subsequently executed by the smart card. Further details of the structure of an exemplary CAP file 16 are discussed in the attached Appendix at pages 53 through 94 and in a publication by Sun Microsystems, Inc. entitled "Java Card Runtime Environment (JCRE) 2.1 Specification," (1998) which is incorporated herein by reference in its entirety.

After conversion, the CAP file 16 can be stored on a computer-readable medium 17 such as a hard drive, a floppy disk, an optical storage medium, a flash device or some other suitable medium.

The CAP file 16 then can be copied or transferred to a terminal 22 (step 66) such as a desktop computer with a peripheral card acceptance device (CAD) 24. In some

embodiments, the terminal 22 can be connected to a network (not shown), such as the Internet, a local area network (LAN) or a wide area network (WAN), which communicates with other computing devices such as a server. In such situations, the CAP file 16 can be accessed and transmitted to the terminal 22 over the network. The CAP file 16 also can be provided to the terminal 22 using a carrier wave, such as a network data transmission.

The CAD 24 allows information to be written to and retrieved from the smart card 40. The CAD 24 includes a card port (not shown) into which the smart card 40 can be inserted. Once inserted, contacts from a connector press against the surface connection area on the smart card 40 to provide power and to permit communications with the smart card, although, in other implementations, contactless communications can be used. The terminal 22 also includes an installation tool 26 which loads the CAP file 16 for transmission to the card 40 (step 68).

The smart card 40 has an input/output (I/O) port 42 which can include a set of contacts through which programs, data and other communications are provided. The card 40 also includes an installation tool 46 for receiving the contents of the CAP file 16 and preparing the applet for execution on the card 40 (step 70). The installation tool 46 can be implemented, for example, as a Java program and can be executed on the card 40. The card 40 also has memory, including volatile memory such as RAM 50. The card 40 also has ROM 52 and non-volatile memory, such as EEPROM 54. The applet prepared by the controller 44 can be stored in the EEPROM 54.

In one particular implementation, the applet is executed by a virtual machine 49 running on a microprocessor 48 (step 72). The virtual machine 49, which can be referred

to as the Java Card™ Virtual Machine, need not load or manipulate the CAP file 16. Rather, the Java Card Virtual Machine 49 executes the applet code previously stored as part of the CAP file 16. The division of functionality between the Java Card Virtual Machine 49 and the installation tool 46 allows both the virtual machine and the installation tool to be kept relatively small.

In general, implementations and applets written for a resource-constrained platform such as the smart card 40 follow the standard rules for Java platform packages. The Java Virtual Machine and the Java programming language are described in T. Lindholm et al., The Java Virtual Machine Specification (1997), and K. Arnold et al., The Java Programming Language Second Edition, (1998), which are incorporated herein by reference in their entirety. Application programming interface (API) classes for the smart card platform can be written as Java source files which include package designations, where a package includes a number of compilation units and has a unique name. Package mechanisms are used to identify and control access to classes, fields and methods. The Java Card API allows applications written for one Java Card-enabled platform to run on any other Java Card-enabled platform. Additionally, the Java Card API is compatible with formal international standards such as ISO 7816, and industry-specific standards such as Europay/MasterCard/Visa (EMV).

The smart card platform of the present invention supports dynamically created objects including both class instances and arrays. A class is implemented as an extension or subclass of a single existing class and its members are methods as well as variables referred to as fields. A method declares executable code that can be invoked and that receives a fixed number of values as

arguments. Classes also can define or implement Java interfaces. An interface is a reference type whose members are constants and abstract methods.

Individual instructions stored in the CAP file 16 and subsequently downloaded to the smart card 40 include an 8-bit operation code (opcode) followed by either zero, one or multiple 8-bit operands (FIG. 3A). Some instructions have no operands and consist only of an opcode. The general form of the inner loop of execution of the Java Card Virtual Machine 49 is illustrated in FIG. 3B. When a method is invoked, the Java Card Virtual Machine 49 allocates a frame which has a set of local variables and contains an operand stack. Many of the operation codes discussed below take one or more values from the operand stack of the current frame, operate on them, and return results to the same stack. The operand stack also is used to pass arguments to methods and receive method results.

Values from the operand stack must be operated upon in ways that are appropriate to their types. The Java Card Virtual Machine 49 supports two kinds of data types: primitive types and reference types. The numeric primitive types supported by the Java Card Virtual Machine 49 are: (1) "byte", whose values are 8-bit signed two's complement integers; (2) "short", whose values are 16-bit signed two's complement integers; and, optionally, (3) "int", whose values are 32-bit signed two's complement integers. The Java Card Virtual Machine 49 also supports a "returnAddress" type, whose values are pointers to the operation codes in the instructions for the virtual machine. The reference types supported by the Java Card Virtual Machine 49 are (1) "class" types; (2) "interface" types; and (3) "array" types. Those reference types are the same as the reference types used in the Java Virtual Machine. The Java

Card Virtual Machine 49 is defined in terms of an abstract storage unit, which can be referred to as a word, which is sufficiently large to hold a value of the type "byte," "short," "reference," or "returnAddress." Two words are sufficiently large to hold a value of the type "int." Multiple-byte operand data is encoded in big-endian order, in other words, with the high-order byte first.

Various keywords, which cannot be used as identifiers or names of declared entities, are supported by the Java Card Virtual Machine 49. A list of the supported keywords is provided in the attached Appendix at page 11. The function and use of those keywords is the same as the corresponding keywords in the Java programming language.

The operation codes which form the executable program stored in the method component 20 of the CAP file 16 are designed to use the same semantics as that used in the class files 10 written in the Java language. Thus, for example, mathematical results and class hierarchies are preserved when the converter 14 transforms the Java class files 10 into the CAP file 16. Nevertheless, as will be evident from the following description, a sequence of instructions that can be executed by the Java Card Virtual Machine 49 differs from programs intended solely to be run by a system incorporating the Java Virtual Machine. Some of the differences are due to the more limited support of data types present in the instruction set discussed below. Other differences result from the fact that the instruction set discussed below is designed to be executable by a virtual machine residing on a resource-constrained device. Some details of the instruction set are intended to optimize the size or performance of either the Java Card Virtual Machine 49 or the programs running on it. Such details include inlining constant pool data directly into the operation

codes or operands, adding multiple versions of a particular instruction to handle different data types, creating composite instructions for operations on the current object, and explicitly handling 16-bit arithmetic.

Referring to FIGS. 4A and 4B, an exemplary instruction set is provided for programs to be executed by the Java Card Virtual Machine 49. Each instruction is identified by a corresponding operation code (opcode) mnemonic and numerical representation. With the exception of two reserved opcodes, impdep1 and impdep2, all of the opcodes typically can be used in a CAP file such as the CAP file 16. The instructions corresponding to the two reserved opcodes provide backdoors or traps to implementation-specific functionality implemented in software and hardware, respectively. Accordingly, the two reserved opcodes typically do not properly appear in the CAP file 16. They are typically used only in representations of programs that were placed on the smart card 40 by means other than receipt of a CAP file.

As previously mentioned, each instruction includes an operation code followed by zero, one, or more operands. In other words, the instructions have the following general format:

operation code
operand1
operand2
. . .

Each word in the instruction format represents a single 8-bit byte or "bytecode." The instruction's opcode is its numeric representation. Each instruction also has a corresponding mnemonic which is its name. However, only the numeric representation is present in the virtual machine code in a CAP file such as the CAP file 36. Detailed

explanations of each instruction including its function and effect on the operand stack appear in the attached Appendix at pages 97 through 215.

Each data manipulation instruction is specific to a particular data type. The instruction set corresponding to the operation codes listed in FIG. 4A supports a subset of the features supported by the Java programming language. By supporting many, although not all, of the features of the Java language and by using the same semantics as the Java class files 10, platform-independent virtual machine code can be written to be executed by the smart card 40 or other resource-constrained device.

As mentioned above, the instruction set for the Java Card Virtual Machine inlines certain data, which would otherwise appear as part of the constant pool 18, directly into the operation codes or operands. Thus, the instruction set itself incorporates certain information that would otherwise be stored in and obtained from a constant pool if one were using the Java class file format. Thus, when the one or more Java class files 10 are converted to the CAP file 16, at least some references to a constant pool are transformed to inline data in the bytecodes associated with the CAP file.

For example, if the virtual machine 49 supports the data type "int," then the "iipush" operation code can be used to push an integer value onto the operand stack. The general format for the "iipush" instruction is illustrated in FIG. 6A, and the format of a corresponding "ldc" instruction from the Java class file format is shown in FIG 6B. The "ldc" instruction includes the operand "index" which is an unsigned byte that is an index into a constant pool. In contrast, the "iipush" instruction, which is executable by the Java Card Virtual Machine 49, eliminates

the need to refer to the constant pool when executing that instruction. Although the "iipush" instruction includes four operands, thereby increasing the length of the instruction, the slightly longer program can be offset by the savings in memory space which is achieved by eliminating the need to store additional information in the constant pool 18.

Similarly, the "checkcast" operation code can be used to check whether an object is of a particular type. The general format for the "checkcast" instruction for the Java Card Virtual Machine 49 is illustrated in FIG. 7A, and the format of a corresponding "checkcast" instruction from the Java class file format is shown in FIG 7B. The data type for the Java Card Virtual Machine 49 has been inlined directly into the instruction, in contrast to the corresponding Java instruction in which the data type is obtained from a constant pool. By inlining some of the information directly into the instruction set, the size of the constant pool 18 that is stored in the CAP file 16 can be reduced.

The foregoing examples illustrate how the instruction set for the Java Card Virtual Machine 49 inlines some information directly into an operand. In some cases, an additional form of inlining is provided by inlining information that would otherwise be stored in the constant pool 18 directly into an operation code. Thus, for example, the instruction set for the Java Card Virtual Machine adds multiple versions of several instruction to handle different data types so that those instructions appear as members of a family of related instructions which share a single description, format and operand stack diagram. Each instruction in such a family of instructions implicitly specifies the data type in the operation code itself. The

table in FIG. 5 provides a list of the data types which are supported by instructions that exist for multiple data types. Wide and composite forms of instructions are not listed. Referring to FIG. 5, a specific instruction, with the data type incorporated into the operation code, is obtained by replacing the "T" in the instruction template in the opcode column by the letter representing the type in the type column. Where the column for a particular instruction is left blank, then no instruction exists supporting the particular operation on that data type. For example, there is a "load" instruction for the data type "short," but there is no "load" instruction for the data type "byte."

With instructions that implicitly incorporate the data type into the operation code, the program can operate more quickly and with less data on the smart card 40 than would otherwise be required. Those advantages arise because the data type is directly encoded in the instructions rather than being obtained from an entry in the constant pool. For example, consider the family of "getfield_T" instructions, which includes the instructions "getfield_a," "getfield_b," "getfield_s" and "getfield_i." The general format of the "getfield_T" instructions for use with the Java Card Virtual Machine 49 is illustrated in FIG. 8A, which contrasts with the format of the corresponding "getfield" instruction in the Java class file format as shown in FIG. 8B. In the instructions for the Java Card Virtual Machine 49 (FIG. 8A), the data type has been inlined not only into the instruction, but it has been inlined directly into the operation code. On the one hand, such features can reduce the amount of information stored in the CAP file 16 and also can reduce the number of operands required for the particular instruction. On the other hand, those features expand the number of distinct operation codes.

Whereas the type of inlining discussed with respect to the "iipush" and "checkcast" opcodes can be advantageous for instructions that tend to be less frequently used, the type of inlining discussed with respect to the "getfield_t" family of instructions can be advantageous particularly for instructions that tend to be used more frequently.

The foregoing examples illustrate how the instruction set for the Java Card Virtual Machine 49 inherently inlines certain information. Another form of inlining information can occur when the CAP file 16 is downloaded to the smart card 40, as explained below.

The installation tool 46 on the smart card 40 can be platform-specific and allows the actual storage of the contents of the CAP file 16 to be determined based on the particular platform receiving and preparing the virtual machine code for execution. Thus, in some implementations, the CAP file 16 may be stored on the smart card 40, or other resource-constrained device, in a manner that differs from the manner in which it was received by the smart card. For example, in some cases, when the CAP file 16 is installed on the card 40, the installation tool 46 can link the contents of the CAP file so that the size of the constant pool 18 can be reduced, and in some cases, so that the constant pool need not be retained or stored on the card. That can be accomplished by converting the constant pool indices that appear as part of the code in the CAP file 18 to the corresponding data at the time of installation, as illustrated in FIGS. 9A and 9B. For example, an index to the constant pool 16 can be replaced by an index to the appropriate field in the object. Thus, the virtual machine code stored on the card 40 will already have the data incorporated within it prior to the time of execution. The virtual machine code, with the constant pool 18 removed,

reduces some of the indirection inherent in a program which uses a constant pool. The amount of memory required to store the bytecodes on the smart card 40 can, therefore, be reduced, and the execution time for the program also can be reduced. Of course, in other implementations, the installation tool 46 may retain the constant pool 18 when the CAP file 16 is downloaded to the smart card 40.

As previously mentioned, the instruction set for the Java Card Virtual Machine also includes composite instructions for performing operations on the current object. In other words, some of the instructions that are executable by the Java Card Virtual Machine 49 allow multiple instructions to be collapsed into a single instruction. In particular, instructions that include a "this" operation, such as the family of "getfield_T_this" instructions and the family of "putfield_T_this" instructions, effectively concatenate multiple instructions. In general, the "this" operation operates on the current object. For example, to fetch a field from the current object, one could use a combination of the "aload_0" instruction and a "getfield_a" instruction as shown in FIG. 10A. Alternatively, one can use the single instruction "getfield_T_this" as illustrated in FIG. 10B. Use of the latter instruction can result in a smaller and faster program code. As previously noted, such features are particularly advantageous in resource-constrained devices such as the smart card 40.

The instruction set for the Java Card Virtual Machine also handles 16-bit arithmetic explicitly. To illustrate how 16-bit arithmetic is handled, consider a situation in which "a," "b" and "c" have been declared as "short" type variables, and the expression "c = (short) a + b;" is to be compiled. The bytecodes written in the Java

class file format are shown in FIG. 11A. As can be seen from FIG. 11A, five opcodes are used to load the values "a" and "b," to add the values "a" and "b," to convert the resulting integer type into a short type, and to store the result. In contrast, only four opcodes are needed to obtain and store the result using the instruction set for the Java Card Virtual Machine 49 which obviates the need to convert the integer type result into a short type. Furthermore, in addition to using fewer bytecodes, the size of the stack can be reduced by as much as fifty percent because the Java Card Virtual Machine operates on 16-bit quantities rather than 32-bit quantities.

An object-oriented, verifiable instruction set is, therefore, provided and allows a file with virtual machine bytecode to be stored on a computer-readable medium. Such a file can be downloaded to the resource-constrained device so that the bytecode can be executed by the resource-constrained device.

Although a virtual machine 49 running on a microprocessor 48 has been described as one implementation for executing the bytecodes on the smart card 40, in alternative implementations, an application-specific integrated circuit (ASIC), or a combination of hardware and firmware can be used as a controller for executing downloaded code instead.

Furthermore, although the invention can be implemented using the operation codes listed in FIGS. 4A and 4B, other operation codes and corresponding instruction sets having certain characteristics are suited for implementing the invention as well. Such characteristics include verifiability, type safety, pointer safety, object-oriented, dynamically linked, virtual machine-based, platform-independence, and use of the same semantics as the Java

language, although not all of those characteristics need to be present in a particular implementation.

As previously discussed, the Java Card instruction set can be used with a variety of different resource-constrained devices, some of which are listed in FIG. 12.

Other implementations are within the scope of the following claims.

What is claimed is:

1. An application software program comprising an object-oriented, verifiable, type-safe and pointer-safe sequence of instructions residing on a computer-readable medium, wherein the program can be loaded to and executed by a resource-constrained device that is based on a processor architecture of fewer than 32 bits.

2. The software program of claim 1 wherein the program can be executed by a resource-constrained device based on a 16-bit processor architecture.

3. The software program of claim 1 wherein the program can be executed by a resource-constrained device based on an 8-bit processor architecture.

4. The software program of claim 1 wherein each instruction includes an 8-bit operation code.

5. The software program of claim 1 wherein the sequence of instructions is hardware platform-independent.

6. The software program of claim 1 wherein the instructions were converted from at least one Java class file and wherein at least some references to a constant pool were transformed to inline data.

7. The software program of claim 6 wherein the instructions comprise operation codes and operands and wherein at least some references to the constant pool are inlined into operands in at least some of the instructions.

8. The software program of claim 6 wherein the instructions comprise operation codes and operands and wherein at least some references to the constant pool are inlined into operation codes in at least some of the instructions.

9. The software program of claim 1 wherein the instructions can be executed by a virtual machine running on a microprocessor residing on the resource-constrained device.

10. The software program of claim 1 wherein the instructions can be executed on a portable smart card.

11. The software program of claim 1 wherein the instructions can be executed by a device that supports multiple data types, wherein the sequence of instructions includes data manipulation instructions, and wherein each data manipulation instruction is specific to a particular data type.

12. The software program of claim 11 wherein the data type associated with each data manipulation instruction is selected from among one of the following types: an 8-bit signed two's complement integer numeric type, a 16-bit signed two's complement integer numeric type and a 32-bit signed two's complement integer numeric type.

13. The software program of claim 11 wherein the instructions can be executed by a device that supports multiple reference types and wherein each reference type is selected from among one of the following types: a class type, an interface type and an array type.

14. The software program of claim 1 wherein the program includes at least one composite instruction for performing an operation on a current object.

15. An application software program comprising an object-oriented, verifiable, type-safe and pointer-safe sequence of instructions residing on a computer-readable medium, wherein the program can be loaded to and executed by a resource-constrained device having random access memory with a capacity of no more than about 64 kilo-bytes.

16. The software program of claim 15 wherein the program can be executed by a resource-constrained device having random access memory with a capacity of no more than about 4 kilo-bytes.

17. The software program of claim 15 wherein each instruction includes an 8-bit operation code.

18. The software program of claim 15 wherein the sequence of instructions is hardware platform-independent.

19. The software program of claim 15 wherein the instructions were converted from at least one Java class file and wherein at least some references to a constant pool were transformed to inline data.

20. The software program of claim 19 wherein the instructions comprise operation codes and operands and wherein at least some references to the constant pool are inlined into operands in at least some of the instructions.

21. The software program of claim 19 wherein the instructions comprise operation codes and operands and wherein at least some references to the constant pool are inlined into operation codes in at least some of the instructions.

22. The software program of claim 15 wherein the instructions can be executed by a virtual machine running on a microprocessor residing on the resource-constrained device.

23. The software program of claim 15 wherein the instructions can be executed on a portable smart card.

24. The software program of claim 15 wherein the instructions can be executed by a device that supports multiple data types, wherein the sequence of instructions includes data manipulation instructions, and wherein each data manipulation instruction is specific to a particular data type.

25. The software program of claim 24 wherein the data type associated with each data manipulation instruction is selected from among one of the following types: an 8-bit signed two's complement integer numeric type, a 16-bit signed two's complement integer numeric type and a 32-bit signed two's complement integer numeric type.

26. The software program of claim 24 wherein the instructions can be executed by a device that supports multiple reference types and wherein each reference type is selected from among one of the following types: a class type, an interface type and an array type.

27. The software program of claim 15 wherein the program includes at least one composite instruction for performing an operation on a current object.

28. A resource-constrained device comprising:

memory for storing an application software program comprising an object-oriented, verifiable, type-safe and pointer-safe sequence of instructions;

random access memory having a capacity of no more than about 64 kilo-bytes; and

a virtual machine implemented on a microprocessor wherein the virtual machine is capable of executing the sequence of instructions.

29. The device of claim 28 wherein the microprocessor is based on an 8-bit architecture.

30. The device of claim 28 wherein the microprocessor is based on a 16-bit architecture.

31. The device of claim 28 wherein each instruction includes an 8-bit operation code.

32. The device of claim 28 wherein the sequence of instructions is hardware platform-independent.

33. The device of claim 28 wherein the instructions were converted from at least one Java class file and wherein at least some references to a constant pool are transformed to inline data.

34. The device of claim 33 wherein the instructions comprise operation codes and operands and wherein at least some references to the constant pool are inlined into operands in at least some of the instructions.

35. The device of claim 33 wherein the instructions comprise operation codes and operands and wherein at least some references to the constant pool are inlined into operation codes in at least some of the instructions.

36. The device of claim 28 wherein the virtual machine supports multiple data types, wherein the sequence of instructions includes data manipulation instructions, and wherein each data manipulation instruction is specific to a particular data type.

37. The device of claim 28 wherein the program includes at least one composite instruction for performing an operation on a current object.

38. A resource-constrained device comprising:

memory for storing an application software program comprising an object-oriented, verifiable, type-safe and pointer-safe sequence of instructions; and

a virtual machine implemented on a microprocessor that is based on an architecture of less than 32 bits, wherein the virtual machine is capable of executing the sequence of instructions.

39. A resource-constrained device comprising:

memory for storing an application software program comprising an object-oriented, verifiable, type-safe and pointer-safe sequence of instructions;

random access memory having a capacity of no more than about 64 kilo-bytes; and

a processor capable of executing the sequence of instructions.

40. The device of claim 39 wherein the processor is based on an 8-bit architecture.

41. The device of claim 39 wherein the processor is based on a 16-bit architecture.

42. A resource-constrained device comprising:

memory for storing an application software program comprising an object-oriented, verifiable, type-safe and pointer-safe sequence of instructions;

random access memory having a capacity of less than about 64 kilo-bytes; and

an application-specific integrated circuit (ASIC) capable of executing the sequence of instructions.

43. The device of claim 42 wherein the ASIC is based on an 8-bit architecture.

44. The device of claim 42 wherein the ASIC is based on a 16-bit architecture.

45. A smart card comprising:

memory for storing an application software program comprising an object-oriented, verifiable, type-safe and pointer-safe sequence of instructions; and

a virtual machine implemented on a microprocessor, wherein the virtual machine is capable of executing the sequence of instructions.

46. The smart card of claim 45 wherein the virtual machine is substantially a Java Card virtual machine.

47. The smart card of claim 45 wherein each instruction includes an 8-bit operation code.

48. The smart card of claim 45 wherein the sequence of instructions is hardware platform-independent.

49. The smart card of claim 45 wherein the instructions were converted from at least one Java class file and wherein at least some references to a constant pool are transformed to inline data.

50. The smart card of claim 45 wherein the instructions comprise operation codes and operands and wherein at least some references to the constant pool are inlined into operands in at least some of the instructions.

51. The smart card of claim 45 wherein the instructions comprise operation codes and operands and wherein at least some references to the constant pool are inlined into operation codes in at least some of the instructions.

52. The smart card of claim 45 wherein the virtual machine supports multiple data types, wherein the sequence of instructions includes data manipulation instructions, and wherein each data manipulation instruction is specific to a particular data type.

53. The smart card of claim 45 wherein the program includes at least one composite instruction for performing an operation on a current object.

54. A method of using an application software program including an object-oriented, verifiable, type-safe and pointer-safe sequence of instructions, the method comprising:

receiving the software program in a resource-constrained device having random access memory with a capacity of no more than about 64 kilo-bytes; and

executing the sequence of instructions on the resource-constrained device.

55. The method of claim 54 further including:

storing the sequence of instructions on the resource-constrained device.

56. The method of claim 54 further including accessing the software program over a computer network prior to downloading the program onto the resource-constrained device.

57. The method of claim 54 further including accessing the software program over the Internet prior to downloading the program onto the resource-constrained device.

58. The method of claim 54 further including:

transforming constant pool indices that appear in the received set of instructions to corresponding data values.

OBJECT-ORIENTED INSTRUCTION SET FOR RESOURCE-CONSTRAINED DEVICES

Abstract

A resource-constrained device such as a smart card or the like includes memory for storing an application software program comprising an object-oriented, verifiable, platform-independent, type-safe and pointer-safe sequence of instructions. The device can also include a virtual machine implemented on a microprocessor where the virtual machine is capable of executing the sequence of instructions. Each instruction includes an operation code, and each data manipulation instruction is specific to a particular data type. The application program can be stored on a computer-readable medium prior to being received by the resource-constrained device. Methods of using such an application program, including accessing the program over the Internet and downloading it to a smart card, also are disclosed.

103040.pal1

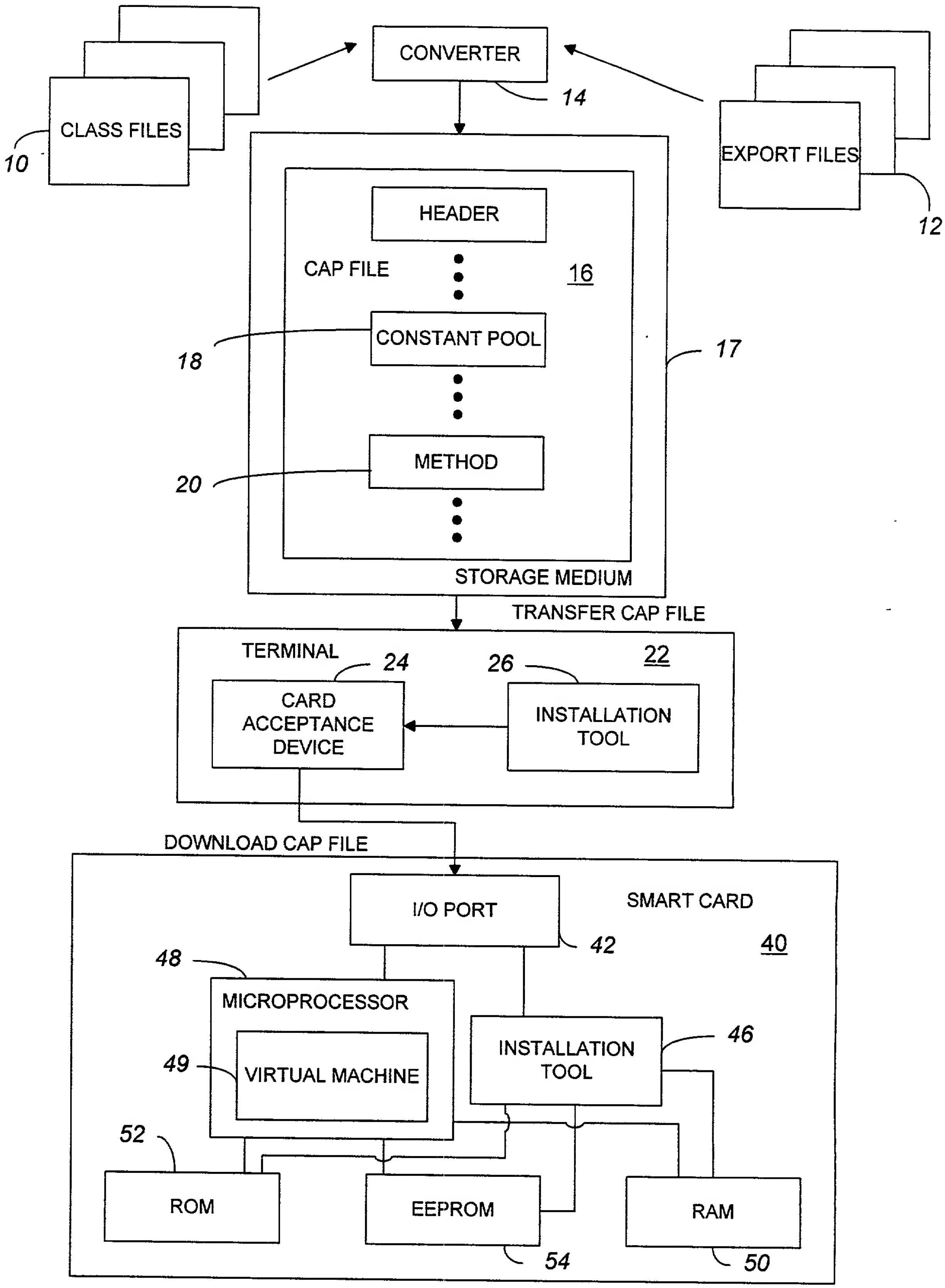
CLASS FILES
10
CONVERTER
14
EXPORT FILES
12
HEADER
CAP FILE
16
CONSTANT POOL
18
17
METHOD
20
STORAGE MEDIUM
TRANSFER CAP FILE
TERMINAL
24
26
22
CARD ACCEPTANCE DEVICE
INSTALLATION TOOL
DOWNLOAD CAP FILE
I/O PORT
SMART CARD
42
40
48
MICROPROCESSOR
49
VIRTUAL MACHINE
INSTALLATION TOOL
46
52
ROM
EEPROM
RAM
54
50

FIG. 1

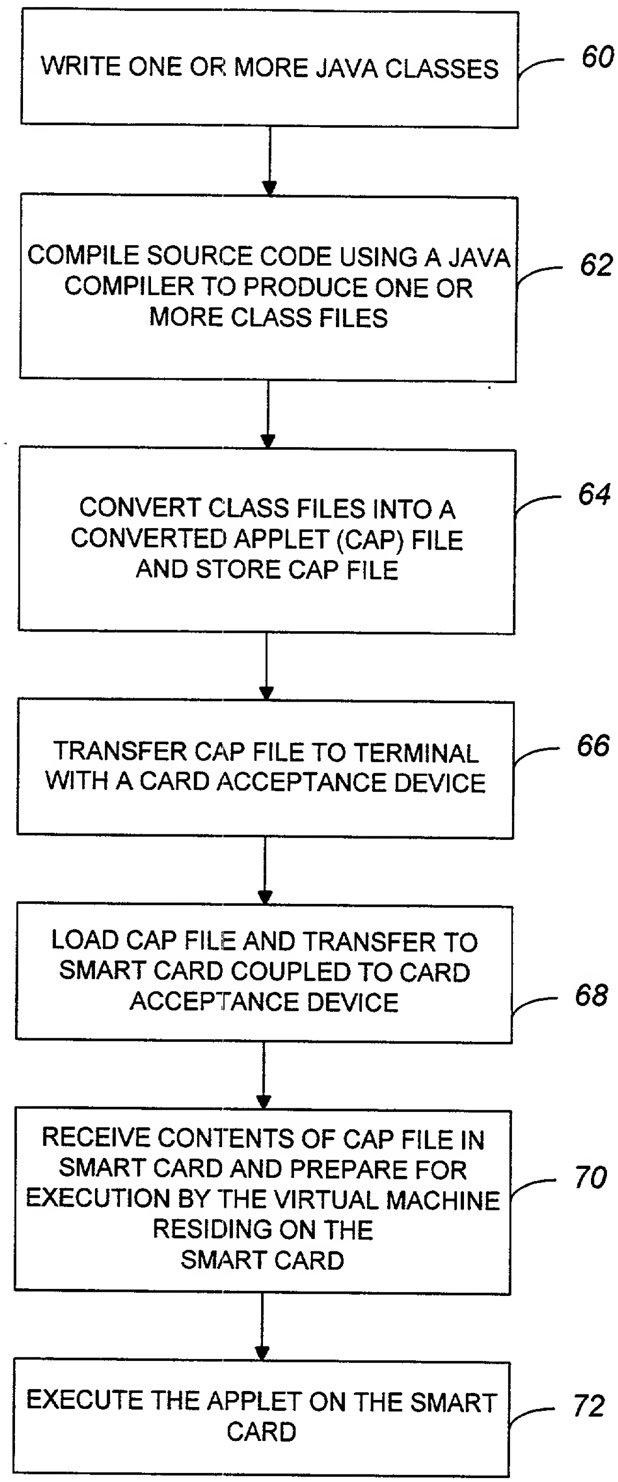
WRITE ONE OR MORE JAVA CLASSES
60
COMPILE SOURCE CODE USING A JAVA COMPILER TO PRODUCE ONE OR MORE CLASS FILES
62
CONVERT CLASS FILES INTO A CONVERTED APPLET (CAP) FILE AND STORE CAP FILE
64
TRANSFER CAP FILE TO TERMINAL WITH A CARD ACCEPTANCE DEVICE
66
LOAD CAP FILE AND TRANSFER TO SMART CARD COUPLED TO CARD ACCEPTANCE DEVICE
68
RECEIVE CONTENTS OF CAP FILE IN SMART CARD AND PREPARE FOR EXECUTION BY THE VIRTUAL MACHINE RESIDING ON THE SMART CARD
70
EXECUTE THE APPLET ON THE SMART CARD
72

FIG. 2

INSTRUCTION { OPCODE
OPERAND
OPERAND
. . .

FIG. 3A

do {

fetch an opcode;
if (operands) fetch operands;
execute the action for the opcode;

} while (there are more opcodes);

FIG. 3B

0	nop
1	aconst_null
2	sconst_m1
3	sconst_0
4	sconst_1
5	sconst_2
6	sconst_3
7	sconst_4
8	sconst_5
9	iconst_m1
10	iconst_0
11	iconst_1
12	iconst_2
13	iconst_3
14	iconst_4
15	iconst_5
16	bspush
17	sspush
18	bipush
19	sipush
20	iipush
21	aload
22	sload
23	iload
24	aload_0
25	aload_1
26	aload_2
27	aload_3
28	sload_0
29	sload_1
30	sload_2
31	sload_3
32	iload_0
33	iload_1
34	iload_2
35	iload_3
36	aaload
37	baload
38	saload
39	iaload
40	astore
41	sstore
42	istore
43	astore_0
44	astore_1
45	astore_2
46	astore_3
47	sstore_0
48	sstore_1
49	sstore_2
50	sstore_3
51	istore_0
52	istore_1
53	istore_2
54	istore_3
55	aastore
56	bastore
57	sastore
58	iastore
59	pop
60	pop2
61	dup
62	dup2
63	dup_x
64	swap_x
65	sadd
66	iadd
67	ssub
68	isub
69	smul
70	imul
71	sdiv
72	idiv
73	srem
74	irem
75	sneg
76	ineg
77	sshl
78	ishl
79	sshr
80	ishr
81	sushr
82	iushr
83	sand
84	iand
85	sor
86	ior
87	sxor
88	ixor
89	sinc
90	iinc
91	s2b
92	s2i
93	i2b
94	i2s
95	icmp
96	ifeq
97	ifne
98	iflt
99	ifge
100	ifgt
101	ifle
102	ifnull
103	ifnonnull
104	if_acmpeq
105	if_acmpne
106	if_scmpeq
107	if_scmpne
108	if_scmplt
109	if_scmpge
110	if_scmpgt
111	if_scmple
112	goto
113	jsr
114	ret
115	stableswitch
116	itableswitch
117	slookupswitch
118	ilookupswitch
119	areturn
120	sreturn
121	ireturn
122	return
123	getstatic_a
124	getstatic_b
125	getstatic_s
126	getstatic_i
127	putstatic_a
128	putstatic_b
129	putstatic_s
130	putstatic_i
131	getfield_a
132	getfield_b
133	getfield_s
134	getfield_i
135	putfield_a
136	putfield_b
137	putfield_s
138	putfield_i
139	invokevirtual
140	invokespecial
141	invokestatic
142	invokeinterface
143	new
144	newarray
145	anewarray
146	arraylength
147	athrow
148	checkcast
149	instanceof
150	sinc_w
151	iinc_w
152	ifeq_w
153	ifne_w
154	iflt_w
155	ifge_w
156	ifgt_w
157	ifle_w
158	ifnull_w
159	ifnonnull_w
160	if_acmpeq_w
161	if_acmpne_w
162	if_scmpeq_w
163	if_scmpne_w
164	if_scmplt_w
165	if_scmpge_w
166	if_scmpgt_w
167	if_scmple_w
168	goto_w
169	getfield_a_w
170	getfield_b_w
171	getfield_s_w
172	getfield_i_w
173	getfield_a_this
174	getfield_b_this
175	getfield_s_this
176	getfield_i_this
177	putfield_a_w
178	putfield_b_w
179	putfield_s_w
180	putfield_i_w
181	putfield_a_this
182	putfield_b_this
183	putfield_s_this
184	putfield_i_this
	...
254	impdep1
255	impdep2

FIG. 4A

aaload	36	iand	84	iload_0	32	putstatic_s	129
aastore	55	iastore	58	iload_1	33	ret	114
aconst_null	1	icmp	95	iload_2	34	return	122
aload	21	iconst_0	10	iload_3	35	s2b	91
aload_0	24	iconst_1	11	ilookupswitch	118	s2i	92
aload_1	25	iconst_2	12	imul	70	sadd	65
aload_2	26	iconst_3	13	ineg	76	saload	38
aload_3	27	iconst_4	14	instanceof	149	sand	83
anewarray	145	iconst_5	15	invokeinterface	142	sastore	57
areturn	119	iconst_m1	9	invokespecial	140	sconst_0	3
arraylength	146	idiv	72	invokestatic	141	sconst_1	4
astore	40	if_acmpeq	104	invokevirtual	139	sconst_2	5
astore_0	43	if_acmpeq_w	160	ior	86	sconst_3	6
astore_1	44	if_acmpne	105	irem	74	sconst_4	7
astore_2	45	if_acmpne_w	161	ireturn	121	sconst_5	8
astore_3	46	if_scmpeq	106	ishl	78	sconst_m1	2
athrow	147	if_scmpeq_w	162	ishr	80	sdiv	71
baload	37	if_scmpge	109	istore	42	sinc	89
bastore	56	if_scmpge_w	165	istore_0	51	sinc_w	150
bipush	18	if_scmpgt	110	istore_1	52	sipush	19
bspush	16	if_scmpgt_w	166	istore_2	53	sload	22
checkcast	148	if_scmple	111	istore_3	54	sload_0	28
dup	61	if_scmple_w	167	isub	68	sload_1	29
dup_x	63	if_scmplt	108	itableswitch	116	sload_2	30
dup2	62	if_scmplt_w	164	iushr	82	sload_3	31
getfield_a	131	if_scmpne	107	ixor	88	slookupswitch	117
getfield_a_this	173	if_scmpne_w	163	jsr	113	smul	69
getfield_a_w	169	ifeq	96	new	143	sneg	75
getfield_b	132	ifeq_w	152	newarray	144	sor	85
getfield_b_this	174	ifge	99	nop	0	srem	73
getfield_b_w	170	ifge_w	155	pop	59	sreturn	120
getfield_i	134	ifgt	100	pop2	60	sshl	77
getfield_i_this	176	ifgt_w	156	putfield_a	135	sshr	79
getfield_i_w	172	ifle	101	putfield_a_this	181	sspush	17
getfield_s	133	ifle_w	157	putfield_a_w	177	sstore	41
getfield_s_this	175	iflt	98	putfield_b	136	sstore_0	47
getfield_s_w	171	iflt_w	154	putfield_b_this	182	sstore_1	48
getstatic_a	123	ifne	97	putfield_b_w	178	sstore_2	49
getstatic_b	124	ifne_w	153	putfield_i	138	sstore_3	50
getstatic_i	126	ifnonnull	103	putfield_i_this	184	ssub	67
getstatic_s	125	ifnonnull_w	159	putfield_i_w	180	stableswitch	115
goto	112	ifnull	102	putfield_s	137	sushr	81
goto_w	168	ifnull_w	158	putfield_s_this	183	swap_x	64
i2b	93	iinc	90	putfield_s_w	179	sxor	87
i2s	94	iinc_w	151	putstatic_a	127		
iadd	66	iipush	20	putstatic_b	128		
iaload	39	iload	23	putstatic_i	130		

FIG. 4B

opcode	byte	short	int	reference
Tspush	bspush	sspush		
Tipush	bipush	sipush	iipush	
Tconst		sconst	iconst	aconst
Tload		sload	iload	aload
Tstore		sstore	istore	astore
Tinc		sinc	iinc	
Taload	baload	saload	iaload	aaload
Tadd		sadd	iadd	
Tsub		ssub	isub	
Tmul		smul	imul	
Tdiv		sdiv	idiv	
Trem		srem	irem	
Tneg		sneg	ineg	
Tshl		sshl	ishl	
Tshr		sshr	ishr	
Tushr		sushr	iushr	
Tand		sand	iand	
Tor		sor	ior	
Txor		sxor	icor	
s2T	s2b		s2i	
i2T	i2b	i2s		
Tcmp			icmp	
if_TcmpOP		if_scmpOP		if_acmpOP
Tlookupswitch		slookupswitch	ilookupswitch	
Ttableswitch		stableswitch	itableswitch	
Treturn		sreturn	ireturn	areturn
getstatic_T	getstatic_b	getstatic_s	getstatic_i	getstatic_a
putstatic_T	putstatic_b	putstatic_s	putstatic_i	putstatic_a
getfield_T	getfield_b	getfield_s	getfield_i	getfield_a
putfield_T	putfield_b	putfield_s	putfield_i	putfield_a

FIG. 5

iipush (Java Card™ Virtual Machine)

Operation: Push integer onto stack

Format:

iipush
byte1
byte2
byte3
byte4

Form: iipush = 20 (0x14)

FIG. 6A

ldc (Java™ Virtual Machine)

Operation: Push item onto stack

Format:

ldc
index

Form: ldc = 18 (0x12)

FIG. 6B

checkcast (Java™ Virtual Machine)

Operation Check whether object is of a given type

Format

checkcast
indexbyte1
indexbyte2

Form checkcast = 192 (0xC0)

FIG. 7A

checkcast (Java Card™ Virtual Machine)

Operation check whether object is of a given type

Format

checkcast
atype
indexbyte1
indexbyte2

Form checkcast = 148 (0x94)

FIG. 7B

getfield _ T (Java Card™ Virtual Machine)

Operation Fetch field from object

Format

getfield _ T
index

Forms

getfield _ a = 131 (0x83)
getfield _ b = 132 (0x84)
getfield _ s = 133 (0x85)
getfield _ i = 134 (0x86)

FIG. 8A

getfield (Java™ Virtual Machine)

Operation Fetch field from object

Format

getfield
indexbyte1
indexbyte2

Form getfield = 180 (0xb4)

FIG. 8B

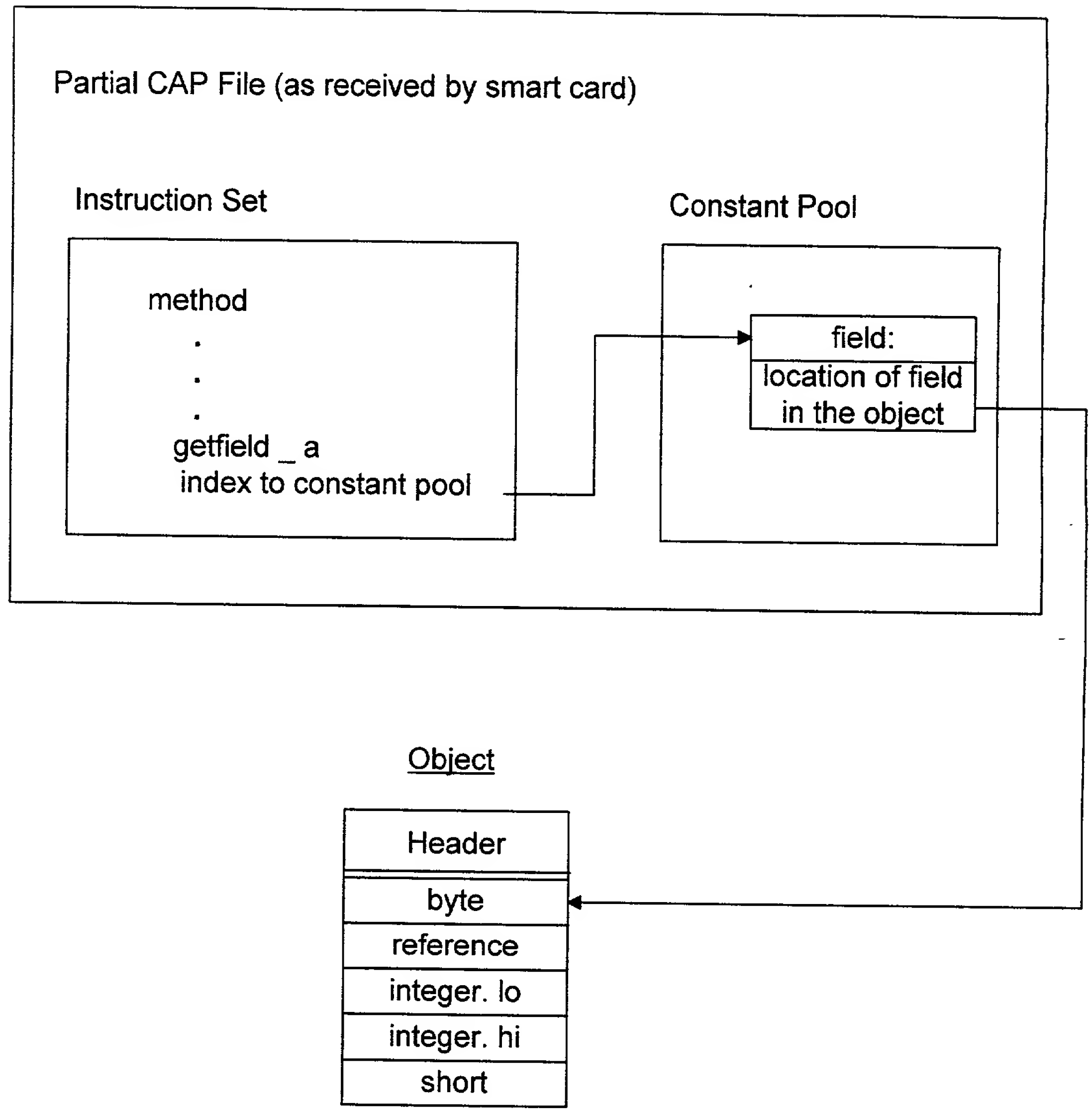
Partial CAP File (as received by smart card)
Instruction Set
method
getfield _ a
index to constant pool
Constant Pool
field:
location of field in the object
Object
Header
byte
reference
integer. lo
integer. hi
short

FIG. 9A

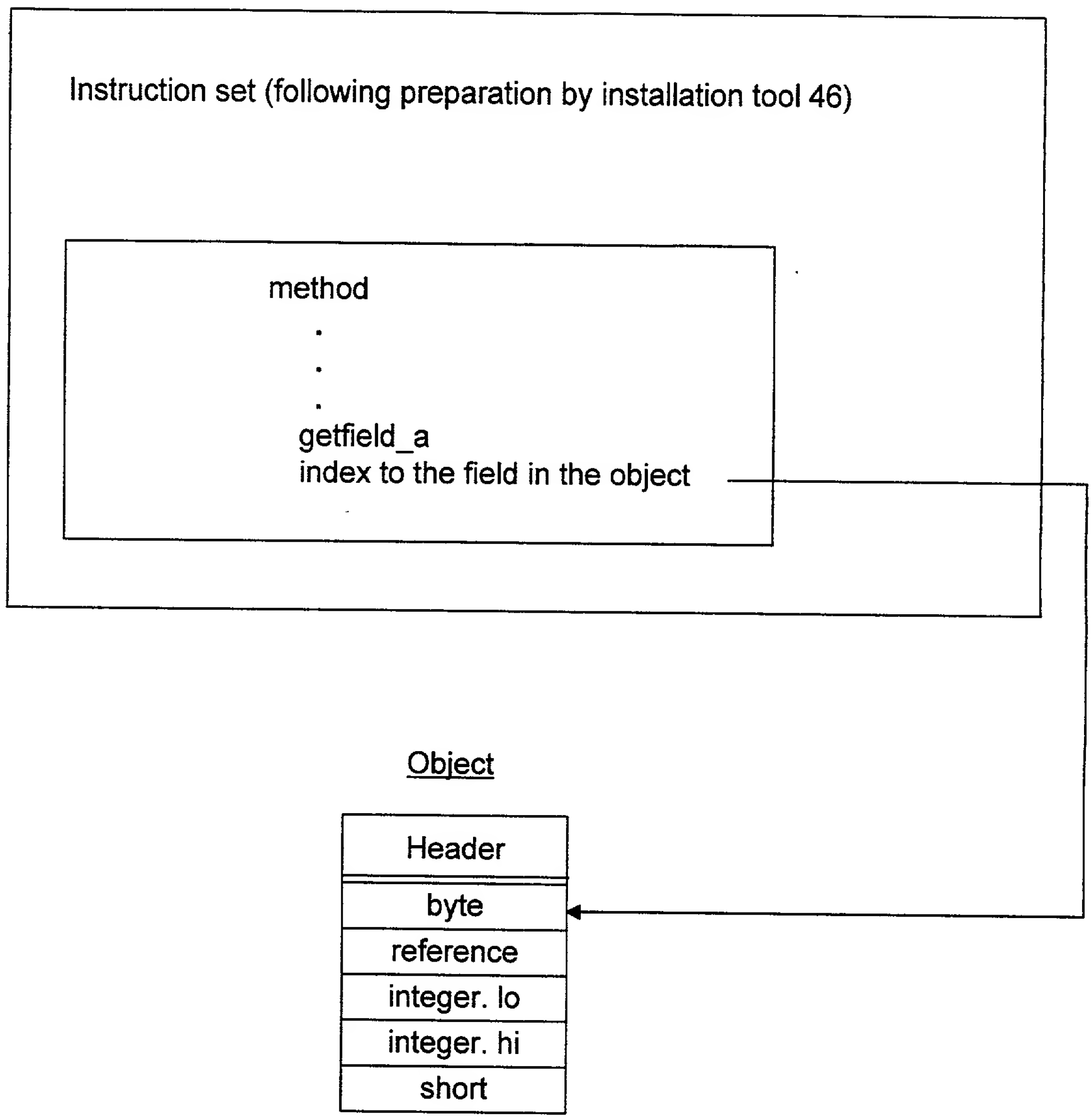
Instruction set (following preparation by installation tool 46)
method
getfield_a
index to the field in the object
Object
Header
byte
reference
integer. lo
integer. hi
short

FIG. 9B

FIG. 10A

```
aload _ 0
getfield _ a
<index>
```

FIG. 10B

```
getfield _ a _ this
<index>
```

```
iload
<a>
iload
<b>
iadd
i2s
istore
```

FIG. 11A

```
sload
<a>
sload
<b>
sadd
sstore
```

FIG. 11B

smart cards

cellular telephones

boundary scan devices

field programmable devices

PDAs

pagers

other small or miniature devices

. . .

FIG. 12

DECLARATION

As a below-named inventor, I hereby declare that:

My residence, post office address and citizenship are as stated next to my name.

I believe I am the original, first and sole inventor (if only one name is listed below) or an original, first and joint inventor (if plural names are listed below) of the subject matter which is claimed and for which a patent is sought on the invention entitled:

OBJECT-ORIENTED INSTRUCTION SET FOR RESOURCE-CONSTRAINED DEVICES

the specification of which:

[X] is attached hereto.

[ ] was filed on ________________________.

[ ] under Serial No. ________________.
[ ] with Express Mail No. ____________________ (Serial Number not yet known).

[ ] was described and claimed in PCT International Application No. _____________________.

I hereby state that I have reviewed and understand the contents of the above-identified specification, including the claims, as amended by any amendment referred to above.

I acknowledge the duty to disclose information which is material to the examination of this application in accordance with Title 37, code of Federal Regulations, Section 1.56(a).

I hereby declare that all statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true; and further that these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code and that such willful false statements may jeopardize the validity of the application or any patent issued thereon.

Full name of inventor: Joshua B. Susser

Inventor's signature ________________________ Date: 2/2/99

Residence: San Francisco, CA
Citizen of: U.S.A.
Post Office Address: 216 Dorland Street, San Francisco, CA 94114

Full name of inventor: Judith E. Schwabe

Inventor's signature ________________________ Date: 2/2/99

Residence: San Mateo, CA
Citizen of: U.S.A.
Post Office Address: 1600 East Third Avenue, #2708, San Mateo, CA 94401